Available online at www.sciencedirect.com
ScienceDirect
Journal of Applied Logic 5 (2007) 545–573
JOURNAL OF APPLIED LOGIC
www.elsevier.com/locate/jal

Disjunctive logic programming with types and objects: The DLV$^+$ system

Francesco Ricca, Nicola Leone *

Department of Mathematics, University of Calabria, Via P. Bucci, cubo 30b, Rende (CS) 87036, Italy

Received 27 October 2005; accepted 13 February 2006
Available online 14 August 2006

Abstract

The paper presents DLV$^+$, a Disjunctive Logic Programming (DLP) system with object-oriented constructs, including classes, objects, (multiple) inheritance, and types. DLV$^+$ is built on top of DLV (a state-of-the art DLP system), and provides a graphical user interface that allows one to specify, update, browse, query, and reason on knowledge bases. Two strong points of the system are the powerful type-checking mechanism and the advanced interface for visual querying.

DLV$^+$ is already used for the development of knowledge based applications for information extraction and text classification.

Keywords: Disjunctive logic programming; Objects; Types; Ontologies; Reasoning

1. Introduction

Disjunctive Logic Programming (DLP) is an advanced formalism for Knowledge Representation and Reasoning (KR&R) [12]. DLP is very expressive in a precise mathematical sense: it is able to express all problems belonging to the complexity class Σ_2^P.[1] Moreover, the availability of a pair of efficient DLP systems, like DLV [19], GnT [16] and, more recently, the disjunctive version of Cmodels [20] make DLP a powerful tool for developing advanced knowledge-based applications [3,22].

The recent application of DLP in the areas of Knowledge Management (KM), Security, and Information Integration [18,24], has confirmed, on the one hand, the viability of the DLP exploitation. On the other hand, it has evidenced some limitations of DLP language and systems. As far as the language is concerned, the need to represent complex real-world entities, like classes, objects, compound objects, and taxonomies, has emerged [24]. Moreover, DLP systems are missing tools for supporting the programmers, like type-checkers and easy-to-use graphical environments, to manage the large and complex domains to be dealt with in real-world applications.

This paper describes the DLV$^+$ system, a first step towards overcoming the above limitations. It is a cross-platform development environment for knowledge modeling and advanced knowledge-based reasoning. The DLV$^+$ system allows for the development of complex applications and allows one to perform advanced reasoning tasks in a user

* Corresponding author.
E-mail addresses: ricca@mat.unical.it (F. Ricca), leone@mat.unical.it (N. Leone).

[1] The class of all decision problems solvable in nondeterministic polynomial time by a Turing machine with an oracle in NP.

1570-8683/$ – see front matter
doi:10.1016/j.jal.2006.02.002

friendly visual environment. The DLV$^+$ system seamlessly integrates the DLV system [19] exploiting the power of a stable and efficient DLP solver.

A strong point of the system is its powerful language, extending DLP by object-oriented features. In particular, the language includes, besides the concept of *relation*, the object-oriented notions of *class, object* (class instance), *object-identity, complex object*, (*multiple*) *inheritance*, and the concept of modular programming by means of *reasoning modules*.

A *class* can be thought of as a collection of individuals that belong together because they share some features. An individual, or *object*, is any identifiable entity in the universe of discourse. Objects, also called class instances, are unambiguously identified by their object-identifier (oid) and belong to a class. A class is defined by a name (which is unique) and an ordered list of attributes, identifying the properties of its instances. Each attribute has a name and a type, which is, in truth, a class. This allows for the specification of *complex objects* (objects made of other objects).

Classes can be organized in a specialization hierarchy (or data-type taxonomy) using the built-in *is-a* relation (*multiple inheritance*).

Relationships among objects are represented by means of *relations*, which, like classes, are defined by a (unique) name and an ordered list of attributes (with name and type).[2]

As in DLP, logic programs are sets of logic rules and constraints. However, DLP$^+$ extends the definition of logic atom by introducing class and relation predicates, and complex terms (allowing for a direct access to object properties). In this way, the DLP$^+$ rules merge, in a simple and natural way, the declarative style of logic programming with the navigational style of the object-oriented systems. In addition, DLP$^+$ logic programs are organized in *reasoning modules*, taking advantage of the benefits of modular programming.

Importantly, the strongly-typed nature of DLP$^+$ allowed for the implementation of a number of *type-checking* routines that verify the correctness of a specification on the fly, resulting in an help for the programmer.

Moreover, DLV$^+$ offers several important facilities driving the development of both the knowledge base and the reasoning modules. Using DLV$^+$, developers and domain experts can create, edit, navigate and query object-oriented knowledge bases by an easy-to-use *visual environment*, enriched by a graphic *query interface* à la QBE.

In short, the contribution of the paper is twofold:

- We define a new language, named DLP$^+$, for Knowledge Representation and Reasoning, extending DLP with relevant constructs of the object-oriented paradigm, like Classes, Types, Objects and Inheritance. We provide a formal definition of both syntax and semantics of DLP$^+$, and illustrate its knowledge modeling features by examples. We analyze the computational complexity of the main decisional problems arising in the context of DLP$^+$.
- We design and implement a system supporting DLP$^+$, named DLV$^+$.
  The system offers all features of DLP$^+$, it provides a user friendly Graphical User Interface, and a powerful type checking mechanism, which supports the user in a fast development of error-free ontologies. DLV$^+$ is endowed also with a visual query interface, allowing to combine navigation and querying for powerful information extraction.
  The system is already employed in practice in a couple of applications for text classification and information extraction (see Conclusion).

The remainder of this paper is structured as follows. In the next section, we present an informal overview of the DLP$^+$ language. After that, we report in Section 3 the formal definition of DLP$^+$ ontologies; while Section 4 gives a formal account of Axioms and Reasoning Modules. Section 6 overviews the architecture and implementation of the DLV$^+$ system. In Section 5 we analyze the complexity of the main decisional problems arising in the context of DLP$^+$. In Section 7 we compare DLP$^+$ with a number of related languages. Finally, Section 8 we draw our conclusions.

[2] Note that, unlike objects, relation instances are not identified by means of oid's.

2. Language overview

The role of a knowledge representation language is to capture domain knowledge and provide a commonly agreed upon understanding of a domain. The specification of a common vocabulary defining the meaning of terms and their relations, usually modeled by using primitives such as concepts organized in taxonomy, relations, and axioms is commonly called an ontology.

In this section we informally describe the DLP$^+$ language, a knowledge representation and reasoning language which allows one to define and to reason on ontologies.

An ontology in DLP$^+$ can be specified by means of *classes*, and *relations*. Classes are organized in an *inheritance* (ISA) hierarchy, while the properties to be respected are expressed through suitable *axioms*, whose satisfaction guarantees the consistency of the ontology. *Reasoning modules* allow us to express rich forms of reasoning on the ontologies.

For a better understanding, we will describe each construct in a separate section and we will exploit an example (the *living being ontology*), which will be built throughout the whole section, thus illustrating the features of the language.

It is worth noting that DLP$^+$ is actually an extension of Disjunctive Logic Programming (DLP—see for instance [12]), which has been enriched by concepts from the object-oriented paradigm; from now on, we assume the reader to be familiar with DLP syntax and semantics (see Appendix A[3]). For a comprehensive introduction to DLP the reader can refer to [10,12].

2.1. Classes

One of the most powerful abstraction mechanism for the representation of a knowledge domain is *classification*, i.e. the process of identifying object categories (*classes*), on the basis of the observation of common properties (*class attributes*).

A *class* can be thought of as a collection of individuals that belong together because they share some properties.

Suppose we want to model the *living being* domain, and we have identified four classes of individuals: *persons*, *animals*, *food*, and *places*. Those classes can be defined in DLP$^+$ as follows:

class *person*.
class *animal*.
class *food*.
class *place*.

The simplest way to declare a class is, hence, to specify the class name, preceded by the keyword **class**. However, when we recognize a class in a knowledge domain, we also identify a number of properties or attributes which are defined for all the individuals belonging to that class.

A class attribute can be specified in DLP$^+$ by means of a pair (*attribute-name* : *attribute-type*), where *attribute-name* is the name of the property and *attribute-type* is the class the attribute belongs to.

For instance, we can enrich the specification of the class *person* by the definition of some properties which are common to each person: the name, age, father, mother, and birthplace.

Note that many properties can be represented by using alphanumeric strings and numbers. To this end, DLP$^+$ features the built-in classes *string* and *integer*, respectively representing the class of all alphanumeric strings and the class of non-negative numbers.

Thus, the class person can be better modeled as follows:

class *person*(
  name : *string*,
  age : *integer*,
  father : *person*,

[3] We actually use DLP with aggregates functions.

mother : *person*,

birthplace : *place*).

Note that this definition is "recursive" (both father and mother are of type *person*). Moreover, the possibility of specifying user-defined classes as attribute types allows for the definition of complex objects, i.e. objects made of other objects. It is worth noting that attributes model the properties that *must* be present in all class instances; properties that *might* be present or not should be modeled, as will be shown later, by using relations.[4]

In the same way, we could enrich the specification of the other above mentioned classes in our domain by adding some attributes. For instance, we could have a name for each *place*, *food* and *animal*, an age for each animal etc.

class *place*(name : string).

class *food*(name : string, origin : place).

class *animal*(name : string, age : integer, speed : integer).

Thus, each class definition contains a set of attributes, which is called *class scheme*. The class scheme represents, somehow, the "structure" of (the data we have about) the individuals belonging to a class.

Next section illustrates how we represent individuals in DLP$^+$.

2.2. Objects

Domains contain individuals which are called *objects* or *instances*.

Each individual in DLP$^+$ belongs to a class and is univocally identified by using a constant called *object identifier* (oid) or *surrogate*.

Objects are declared by asserting a special kind of logic facts (asserting that a given instance belongs to a class). For example, we declare that "Rome" is an instance of the class *place* as follows:

rome : *place*(*name* : "*Rome*").

Note that, when we declare an instance, we immediately give an oid to the instance (in this case is *rome*), and a value to the attributes (in this case the *name* is the string "Rome").

The oid *rome* can now be used to refer to that place (e.g. when we have to fill an attribute of another object). Suppose that, in the *living being* domain, there is a person (i.e. an instance of the class person) whose name is "John". John is 34 years old, lives in Rome, his father and his mother are identified by *jack* and *ann* respectively. This instance can be declared as follows:

john : *person*(*name* : "*John*", *age* : 34, *father* : *jack*, *mother* : *ann*, *birthplace* : *rome*).

In this case, "*john*" is *the object identifier* of this instance, while "*jack*", "*ann*", and "*rome*" are suitable oids respectively filling the attributes *father*, *mother* (both of type person) and *birthplace* (of type place).

The language semantics (and our implementation) guarantees the referential integrity, both *jack*, *ann* and *rome* have to exist when *john* is declared.

2.3. Inheritance

Another relevant abstraction tool in the field of knowledge representation is the *specialization/generalization* mechanism, allowing to organize concepts of a knowledge domain in a taxonomy. This is obtained in the object-oriented languages by using the well-known mechanism of inheritance.

Inheritance is supported by DLP$^+$, and class hierarchies can be specified by using the special binary relation *isa*.

For instance, one can exploit inheritance to represent some special categories of persons, like *students* and *employees*, having some extra attribute, like a school, a company etc. This can be done in DLP$^+$ as follows:

class *student* ***isa*** {*person*}(

[4] In other words, an attribute $(n : k)$ of a class c is a total function from c to k; while partial functions from c to k can be represented by a binary relation on (c, k).

code : *string*,

school : *string*,

tutor : *person*).

class *employee* ***isa*** {*person*}(

salary : *integer*,

skill : *string*,

company : *string*,

tutor : *employee*).

In this case, we have that *person* is a more generic concept or *superclass* and both *student* and *employee* are a specialization (or *subclass*) of *person*. Moreover, an instance of *student* will have both the attributes: code, school, and tutor, which are defined locally, and the attributes: name, age, father, mother, and birthplace, which are defined in *person*. We say that the latter are "inherited" from the superclass *person*. An analogous consideration can be made for the attributes of *employee* which will be name, age, father, mother, birthplace, salary, skill, company, and tutor.

An important (and useful) consequence of this declaration is that each proper instance of both *employee* and *student* will also be automatically considered an instance of *person* (the opposite does not hold!).

For example, consider the following two instances of *student* and *employee*:

al : *student*(*name* : "*Alfred*", *age* : 20, *father* : *jack*, *mother* : *betty*, *birthplace* : *rome*,

code : "100", *school* : "*Cambridge*", *tutor* : *hanna*).

jack : *employee*(*name* : "*Jack*", *age* : 54, *father* : *jim*, *mother* : *mary*, *birthplace* : *rome*,

salary : 1000, *skill* : "*Java programmer*", *company* : "*SUN*", *tutor* : *betty*).

They are automatically considered also instances of person as follows:

al : *person*(*name* : "*Alfred*", *age* : 20, *father* : *jack*, *mother* : *betty*, *birthplace* : *rome*).

jack : *person*(*name* : "*Jack*", *age* : 54, *father* : *jim*, *mother* : *mary*, *birthplace* : *rome*).

Note that it is not necessary to assert the above two instances, both *al* and *jack* are automatically considered instances of *person*.

In DLP$^+$ there is no limitation on the number of superclasses (i.e. multiple inheritance is allowed). Thus, a class can be a specialization of any number of classes, and, consequently, it inherits all the attributes of its superclasses.

As an example, consider the following declaration:

class *stud-emp* ***isa*** {student, employee}(

workload : *integer*).

So, the class *stud-emp* (exploiting multiple inheritance) is a subclass of both *student* and *employee*. Note that, the attribute tutor is defined in both *student*, with type *student*, and *employee* with type *employee*.[5]

In this case, the attribute *tutor* will be taken only once in the scheme of *stud_emp*, but it is not intuitive what type will be taken for it.

This tricky situation is dealt with by applying a simple criterion. The type of the "conflicting" attribute *tutor* will be *employee*, which is the "intersection" (somehow in the sense of instance sharing) of the two types of the tutor attribute (*person* and *employee*). This choice is reasonably safe, and guarantees that all instances of *stud_emp* are correct instances of both *student* and *employee* (see Section 3).

We complete the description of inheritance recalling that there is also another built-in class in DLP$^+$, which is the superclass of all the other classes and is called object (or $\top$).

For a formal description of inheritance we refer the reader to Section 3.

[5] We acknowledge that is quite unnatural that the tutor of a student employee is an employee. Actually we made this choice to show an important feature of the language.

2.4. Relations

A fundamental feature of a knowledge representation language is the capability to express relationships among the objects of a domain. This can be done in DLP$^+$ by means of *Relations*.

Relations are declared like classes: the keyword **relation** (instead of **class**) precedes a list of attributes.

As an example, the relation *friend*, which models the friendship between two persons, and the relation *lived* containing information about the places where a person lived can be declared as follows:

relation *friend*(*pers*1 : *person*, *pers*2 : *person*).

relation *lived*(*per* : *person*, *pla* : *place*, *period* : *string*).

Like classes, the set of attributes of a relation is called *scheme* while the cardinality of the scheme is called arity. The scheme of a relation defines the structure of its tuples (this term is borrowed from database terminology).

In particular, to assert that a person, say "john", lived in Rome for two years we write the following logic fact:

lived(*per* : *john*, *pla* : *rome*, *period* : "*two years*").

We call this assertion a tuple of the relation *lived*. Thus, tuples of a relation are specified similarly to class instances, that is, by asserting a set of facts (but tuples are not equipped with an oid).

2.5. Axioms and consistency

The structural representation of a knowledge domain is obtained in DLP$^+$ by specifying classes and relations. In general, this information is not enough to obtain a correct description of the domain. Often, it is necessary to impose constraints asserting additional conditions which hold in the domain.

These assertions are modeled in DLP$^+$ by means of *axioms*.

An *axiom* is a consistency-control construct modeling sentences that are always true (at least, if everything we specified is correct). They can be used for several purposes, such as constraining the information contained in the ontology and verifying its correctness.

As an example suppose we declared the relation colleague, which associates persons working together in a company, as follows:

relation *colleague* (*emp*1 : *employee*, *emp*2 : *employee*).

It is clear that the information about the company of an employee (recall that there is an attribute company in the scheme of the class employee) must be consistent with the information contained in the tuples of the relation colleague. To enforce this property we assert the following axioms:

(1) :– *colleague*(*emp*1 : *X*1, *emp*2 : *X*2), `not` *colleague*(*emp*1 : *X*2, *emp*2 : *X*1).

(2) :– *colleague*(*emp*1 : *X*1, *emp*2 : *X*2),

*X*1 : *employee*(*company* : *C*), `not` *X*2 : *employee*(*company* : *C*).

The above axioms states that, (1) the relation colleague is symmetric, and (2) if two persons are colleagues and the first one works for a company, then also the second one works for the same company.

Note that DLP$^+$ axioms do not derive new knowledge, but they are only used to model sentences that must be always true, like integrity constraints.[6]

The usefulness of axioms is rather clear, as they allows one to enforce the consistency of the specified ontology.

Consequently, if an axiom is violated, then we say that the ontology is inconsistent (that is, it contains information which is, somehow, contradictory or not compliant with the intended perception of the domain).

[6] The difference between axioms and constraints is that axioms are specifically conceived to work with the knowledge contained in an ontology, while constraints are conceived in order to enforce some property in a logic program.

2.6. Reasoning modules

Given an ontology, it can be very useful to reason about the data it describes.

Reasoning modules are the language components endowing DLP$^+$ with powerful reasoning capabilities. Basically, a *reasoning module* is a disjunctive logic program conceived to reason about the data described in an ontology. Reasoning modules in DLP$^+$ are identified by a name and are defined by a set of (possibly disjunctive) logic rules and integrity constraints.

Syntactically, the name of the module is preceded by the keyword *module* while the logic rules are enclosed in curly brackets (this allows one to collect all the rules constituting the encoding of a problem in a unique definition identified by a name). Moreover, it is possible to define *derived* predicates having a "local scope" without giving a scheme definition. This gives the possibility to exploit a form of modular programming, because it becomes possible to organize logic programs in a simple kind of library.

As an example consider the following module, which allows to single out in the derived predicate *youngAndShy* the names of the persons who are less than 18 years old, and who have less than ten friends:

$$\begin{aligned}&\textbf{\textit{module}}(\mathit{shyFriends})\{\\ &\quad \mathit{youngAndShy}(N) \ :\!-\ P : \mathit{person}(\mathit{name} : N, \mathit{age} : A), A < 18,\\ &\qquad\qquad\qquad\qquad \#\mathit{count}\{F : \mathit{friend}(\mathit{pers}1 : P, \mathit{pers}2 : F)\} < 10.\\ &\}\end{aligned}$$

Note that, this information is implicitly present in the ontology, and the reasoning module just allows to make it explicit.

We now show another example demonstrating that the reasoning power of DLP$^+$ can be exploited also for solving complex real-world problems.

Given our living being ontology, we want to compute a project team satisfying the following restrictions (i.e. we want to solve an instance of *team building problem*):

- the project team has to be constituted of a fixed number of employees;
- the availability of a given number of different skills has to be ensured inside the team;
- the sum of the salaries of the team members cannot exceed a given budget;
- the salary of each employee in the team cannot exceed a certain value.

Suppose that the ontology contains the class *project* whose instances specify the information about the project requirements, i.e. the number of team employees, the number of different skills required in the project, the available budget, the maximum salary of each team employee:

$$\textbf{\textit{class}}\ \mathit{project}(\mathit{numEmp} : \mathit{integer}, \mathit{numSk} : \mathit{integer}, \mathit{budget} : \mathit{integer}, \mathit{maxSal} : \mathit{integer}).$$

We can solve the above team building problem with the following module:

$$\begin{aligned}&\textbf{\textit{module}}(\mathit{teamBuilding})\{\\ &(r) \quad \mathit{inTeam}(E, P) \vee \mathit{outTeam}(E, P) \ :\!-\ E : \mathit{employee}(),\ P : \mathit{project}().\\ &(c_1) \ :\!-\ P : \mathit{project}(\mathit{numEmp} : N),\ \texttt{not}\ \#\texttt{count}\{E : \mathit{inTeam}(E, P)\} = N.\\ &(c_2) \ :\!-\ P : \mathit{project}(\mathit{numSk} : S),\ \texttt{not}\ \#\texttt{count}\{\mathit{Sk} : E : \mathit{employee}(\mathit{skill} : \mathit{Sk}), \mathit{inTeam}(E, P)\} \geqslant S.\\ &(c_3) \ :\!-\ P : \mathit{project}(\mathit{budget} : B),\ \texttt{not}\ \#\texttt{sum}\{\mathit{Sa}, E : E : \mathit{employee}(\mathit{salary} : \mathit{Sa}), \mathit{inTeam}(E, P)\} \leqslant B.\\ &(c_4) \ :\!-\ P : \mathit{project}(\mathit{maxSal} : M),\ \texttt{not}\ \#\texttt{max}\{\mathit{Sa} : E : \mathit{employee}(\mathit{salary} : \mathit{Sa}), \mathit{inTeam}(E, P)\} \leqslant M.\\ &\}\end{aligned}$$

Intuitively, the disjunctive rule r guesses whether an employee is included in the team or not, generating the search space, while the constraints c_1, c_2, c_3, and c_4 model the project requirements, cutting off the solutions that do not satisfy the constraints.

Concluding, reasoning modules isolate a set of logic rules and constraints conceptually related, they exploit the expressive power of disjunctive logic programming allowing to perform complex reasoning tasks on the information encoded in an ontology.

2.7. Querying

An important feature of the language is the possibility of asking queries in order to extract knowledge contained in the ontology, but not directly expressed. As in DLP a query can be expressed by a conjunction of atoms, which, in DLP$^+$, can also contain complex terms.

As an example, we can ask for the list of persons having a father who is born in Rome as follows:

$X : person(father : person(birthplace : place(name : \text{“}Rome\text{”})))?$

Note that we are not obliged to specify all attributes; rather we can indicate only the relevant ones for querying. In general, we can use in a query both the predicates defined in the ontology and the derived predicates in the reasoning modules.

For instance, consider the reasoning module *shyFriends* defined in the previous section, the following two queries ask, respectively: (i) whether the number of people who are “young and shy” and were born in Rome is less than ten, and whether there is a person whose name is “Jack” and is “young and shy”:

$\#count\{X : youngAndShy(X), X : person(birthplace : rome)\} = 10?$

$youngAndShy(person(name : \text{“}Jack\text{”}))?$

3. DLP$^+$ ontologies

In this section we give a formal definition of DLP$^+$ ontologies.

We start by defining the structural information of the ontology: the schemes of classes and relations, along with the inheritance (ISA) taxonomy. We then define the instances of both classes and relations (the class instances are objects, while the relation instances are tuples, representing relationships among objects). The definition of classes and relations includes both structural information on the schemes and the extensions (sets of instances). The notion of Ontology is finally presented.

3.1. Names

Let $E = K \cup R$ be a finite set of *entity* names. The sets K and R are disjoint, and their elements are called *class names* and *relation names*, respectively. We require that E contains alphanumeric strings beginning with a lowercase letter.

The set K also contains three special elements, namely, object (also denoted by $\top$), string, and integer, called *built-in class names*.

Let A be a finite set of symbols called *attribute* names.

Let D be an infinite set of constants, called *object identifiers* (oid's).

3.2. Class and relation schemes

The structure of an entity is defined in DLP$^+$ by a *scheme*. In order to define what a scheme is, we first need to introduce the concept of *attribute*.

Definition 1. An attribute α is a pair $(a : c)$, where $a \in A$ is an attribute name, and $c \in K$ is a class name. In particular, a and c are called the *name* and the *type* of α.

Example 1. As an example, the pair $(address : string)$ denotes an attribute of name “address” and of type *string*.

Definition 2. A class scheme Σ_c is a triple $\Sigma_c = \langle c, A_c, S_c \rangle$, where:

- $c \in K$ is a class name.
- A_c is a set of attributes such that no two distinct elements share the same attribute name.
- S_c is a set of class names.

We say that k is a direct *superclass* of c if $k \in S_c$, and, we say that S_c is the set of the direct *superclasses* of c.
We require that the following three conditions hold:

(i) $A_k = \emptyset$, for each $k \in \{object, string, integer\}$
(ii) $S_{string} = S_{integer} = \{object\}$

Condition (i) requires that built-in classes do not have attributes; condition (ii) requires that *string* and *integer* have only *object* as superclass.

A class scheme $\langle c, \{a_1, \ldots, a_n\}, \{s_1, \ldots, s_n\}\rangle$ for a class c is syntactically declared as follows:

class c *isa*$\{s_1, \ldots, s_n\}$ $(a_1, \ldots, a_n)$.

If c has no direct superclass apart from *object*, then the syntax is simplified to:

class$(a_1, \ldots, a_n)$.

In an analogous way, we define the scheme for a relation.

Definition 3. A relation scheme Σ_r is a pair $\Sigma_r = \langle r, A_r\rangle$, where:

- $r \in R$ is a relation name
- A_r is a set of attributes such that no two distinct elements share the same attribute name.

A relation scheme $\langle r, \{a_1, \ldots, a_n\}\rangle$, is syntactically defined as follows:

relation $r(a_1, \ldots, a_n)$.

The cardinality of the scheme for a relation r is called *arity* of r.

Example 2. Let *place, person, student, employee* and *stud-emp* be the non built-in class names in the set K. Further, let *lived* and *friend* be the relation names in R. Now a possible specification for their schemes is the following:

class *place*(*name* : *string*).
class *person*(
name : *string*,
age : *integer*,
father : *person*,
mother : *person*,
birthplace : *place*).
class *student* ***isa*** {person}(
code : *string*,
school : *string*,
tutor : *person*).
class *employee* ***isa*** {person}(
salary : *integer*,
skill : *string*,
company : *string*,
tutor : *employee*).

class *stud-emp* ***isa*** {student, employee}(

workload : *integer*).

relation *friend*(*pers*1 : *person*, *pers*2 : *person*).

relation *lived*(*per* : *person*, *pla* : *place*, *period* : *string*).

Note that the arity of *friend* is 2, and the arity of *lived* is 3.

3.3. Scheme closure (attributes inheritance)

In order to deal with inheritance, we now define the notion of scheme closure, which allows us to extend the notion of scheme to encompass those attributes that are inherited from the schemes of its superclasses.

Let the function $isa : K \to 2^K$ be defined such that $isa(k) = S_k$, and let isa^* be the reflexive-transitive closure of *isa*. We assume that $\langle K, isa^* \rangle$ is a semi lattice where object is the upper bound of K. Note that, the built-in classes string and integer are comparable only with object and themselves (in the isa^* relation).

Given two class names $c, k \in K$ if $c\ isa^*\ k$, we say that k is a *superclass* of c and c is a *sub-class* of k.[7]

Given a class name c, let $A_c^+ = \bigcup_{c\ isa*\ k} \Sigma_k$ (i.e. A_c^+ is the set of attributes occurring either in the scheme of c or in the scheme of any of its superclasses).

Thus we define the scheme closure σ_c of c in such a way that if two attributes in A_c^+, say a_1 and a_2 have the same attribute name, σ_c will contain only one element with that attribute name, whose type will be the meet of types of a_1 and a_2.

Definition 4. Let $A_c^* = \{(a : v) \mid (a : x) \in A_c^+ \wedge v = glb(\{y \mid (a : y) \in A_c^+\})\}$.[8] The scheme closure of c, named σ_c, is the pair $\sigma_c = \langle c, A_c^* \rangle$.

The cardinality of the scheme closure for a class c is called *arity* of c.

Example 3. The scheme closures of the not built-in classes defined in Example 2 are shown next:

$\sigma_{place} = \{(name : string)\}$

$\sigma_{person} = \{(name : string), (age : integer), (father : person),$
$(mother : person), (birthplace : place)\}$

$\sigma_{student} = \{(name : string), (age : integer), (father : person),$
$(mother : person), (birthplace : place), (code : string),$
$(school : string), (tutor : person)\}$

$\sigma_{employee} = \{(name : string), (age : integer), (father : person),$
$(mother : person), (birthplace : place), (salary : integer),$
$(skill : string), (company : string), (tutor : employee)\}$

$\sigma_{stud\text{-}emp} = \{(name : string), (age : integer), (father : person),$
$(mother : person), (birthplace : place), (code : string),$
$(school : string), (salary : integer), (skill : string),$
$(company : string), (workload : integer), (tutor : employee)\}.$

For *place* and *person* the attributes in their scheme closure are exactly those in the corresponding schemes, because the respective sets of direct superclasses contain only *object*. The scheme closure of *student* and *employee* corresponds

[7] From the definition of isa^* it follows that if c is *direct* superclass of k then c is also superclass of k and vice versa.

[8] $glb(\{y \mid (a : y) \in A_c^+\})$ denotes the greatest lower bound of the types of the attributes with name a in A_c^+. If no lower bound exists then the scheme is not well-formed and the system prompts an error.

to the union of the attributes appearing in the class scheme of *person* with the attributes appearing in its own scheme, respectively. The scheme closure of *stud-emp* is obtained by taking the attributes appearing in the scheme closure of *student* or *employee* where the "conflicting" attribute *tutor* has type *employee* (which is the glb of person and employee). Moreover, the arity of *place, person, student, employee* and *stud-emp* is, respectively, 1, 5, 8, 9, 12.

3.4. Class and relation instances

In this section we provide the formal definition of *class instance* and *relation instance* (or *tuple*).

Definition 5. An instance of a class c is a triple $\langle c, oid, T_c \rangle$, where $oid \in D$ is an object identifier, and T_c is a subset of $A \times D$ such that, for each attribute $(a : k) \in \sigma_c$ there is exactly one element $(a : z) \in T_c$. We call z the value of a.

An instance of a class c, having closure $\sigma_c = \{a_1 : c_1, \ldots, a_n : c_n\}$, is syntactically defined as follows:

$$oid : c(a_1 : k_1, \ldots, a_n : k_n).$$

where $k_i \in D$ for each $i \in [1, n]$.

Example 4. Consider the class *student* of the previous example, its scheme closure is

$$\sigma_{student} = \{(name : string), (age : integer), (father : person), (mother : person),$$
$$(birthplace : place), (code : string), (school : string), (tutor : person)\}.$$

The following instance:

$$I = \langle student, al, \{(name : \text{“}Alfred\text{”}), (age : 20), (father : jack), (mother : betty),$$
$$(birthplace : rome), (code : \text{“}100\text{”}), (school : \text{“}Cambridge\text{”}), (tutor : hanna)\}\rangle$$

can be declared in DLP$^+$ as:

$$al : student(name : \text{“}Alfred\text{”}, age : 20, father : jack, mother : betty, birthplace : rome,$$
$$code : \text{“}100\text{”}, school : \text{“}Cambridge\text{”}, tutor : hanna).$$

In a similar way, we now define the concept of tuple.

Definition 6. A tuple of a relation r is a pair $\langle r, T_r \rangle$, where T_r is a subset of $A \times D$ such that, for each attribute $(a : k)$ in the scheme of r there exists exactly one element $(a : z) \in T_r$.

Analogously, a tuple of relation r having scheme $\Sigma_r = \langle r, \{(a_1 : c_1), \ldots, (a_n : c_n)\}\rangle$ is syntactically defined as follows:

$$r(a_1 : k_1, \ldots, a_n : k_n).$$

where $k_i \in D$ for each $i \in [1, n]$.

Example 5. Consider the following tuple of the relation *lived* (the *lived* relation was defined in Example 2):

$$T = \langle lived, \{(per : john), (pla : rome), (period : \text{“}two\ years\text{”})\}\rangle$$

can be declared in DLP$^+$ as:

$$lived(per : john, pla : rome, period : \text{“}two\ years\text{”}).$$

3.5. Admissible ontologies

We are now in the position to define ontologies.

Definition 7. An *ontology* $\mathcal{O}$ is a tuple $\mathcal{O} = \langle \mathcal{C}_s, \mathcal{R}_s, \mathcal{C}_i, \mathcal{R}_i \rangle$, where $\mathcal{C}_s$ and $\mathcal{R}_s$ are, respectively, the set of class and relation schemes, while $\mathcal{C}_i$ and $\mathcal{R}_i$ are, respectively, the set of class instances and the set of relation tuples (also called, respectively, classes and relations extensions).

We impose that a DLP$^+$ ontology satisfies some admissibility conditions to guarantee: object identity, instance inheritance, typing, and referential integrity.

Condition 1 *(Object identity and instance inheritance).*

- (i) if k and c are two class names such that k *isa** c holds, then for each instance $\langle c, oid, T_c \rangle$ of c there exists an instance $\langle k, oid, T_k \rangle$ of k such that $T_c \subseteq T_k$.
- (ii) if $\langle c, oid, T_c \rangle$ is an instance of class c, and $\langle k, oid, T_k \rangle$ is an instance of class k then either c *isa** k or k *isa** c.

Condition 2 *(Typing and referential integrity).*

- Let $oid : c(a_1 : v_1, \ldots, a_n : v_n)$ be an instance of a class $c \in \mathcal{C}_i$, and let $\sigma_c = \{(a_1 : t_1), \ldots, (a_n : t_n)\}$ be the closure of c, then for each $j \in [1, n]$: (i) there exists an instance $I_j \in \mathcal{C}_i$ having v_j as the object identifier, (ii) I_j is an instance of the class t_j.
- Let $r(a_1 : v_1, \ldots, a_n : v_n)$ be a tuple of a relation $r \in \mathcal{R}_i$, and let $\Sigma_r = \langle r, \{(a_1 : t_1), \ldots, (a_n : t_n)\} \rangle$ be the scheme of r, then for each $j \in [1, n]$: (i) there exists an instance $I_j \in \mathcal{C}_i$ having v_j as the object identifier, (ii) I_j is an instance of the class t_j.

Intuitively, Condition 1 ensures that (i) superclasses must include all the instances of their subclasses (instance inheritance), and that (ii) two instances may share the same *oid* only if they represent the same object seen as the instance of different classes related by *isa** (object identity); Condition 2 ensures that, for both class and relation instances, the attribute values are object identifiers of some instance (referential integrity) belonging to the class which is the type of the attribute (or consistent typing).

In order to simplify the specification of an ontology in our system, we allow that the user declares only the so-called *proper* instances of a class (i.e. the instances which do not belong to a sub-class),[9] and accordingly with Condition 1, the system will build the complete extension of the class.

4. Consistency and reasoning

An important feature of the DLP$^+$ language is its capability to reason on the knowledge contained in an ontology. In this section we introduce three important constructs of the DLP$^+$ language: *Axioms, Reasoning modules, and Queries*. Moreover, we formally define the notion of consistent ontology, and we give the semantics of reasoning modules formally.

4.1. Syntax of axioms, reasoning modules, and queries

We first define a logic language which extends DLP in order to deal with ontologies.

Let V be a finite set of symbols denoting *variables*. The elements of V are alphanumeric strings beginning with an uppercase letter. Moreover, V contains the special symbol "_" that identifies the anonymous variable.

Let M be a set of symbols representing *reasoning module names*.

[9] More formally, given an ontology $\mathcal{O}$, the proper instances of a class c are the instances $\langle c, oid, T_c \rangle \in \mathcal{C}_i$ such that there exists no $\langle k, oid, T_k \rangle \in \mathcal{C}_i$ such that $k \neq c$.

Let Q be a set of derived predicate symbols.

We require that Q contains alphanumeric strings beginning with a lowercase letter, and, moreover, we require that Q and E (the set of entity names), are disjoint.

Definition 8. A *term* is inductively defined as follows:

- constants in D and variables in V are terms (*simple terms*).
- an expression of the form $q(a_1 : t_1, \ldots, a_n : t_n)$ is a term if: (i) $t_i, \ldots, t_n$ are terms; (ii) $q \in K$ is a class name; (iii) the closure of q is $\sigma_q = \{a_1 : c_1, \ldots, a_m : c_m\}$; (iv) $n \leqslant m$ (i.e. n is less than or equal to the arity of q).

We say that a term is *ground* if it does not contain variables.

Definition 9. An *atom* is either a *class atom* or a *relation atom* or a *built-in atom* or an *derived atom*, or an *aggregate atom*, where:

- a *class atom* is an expression $oid : c(a_1 : t_1, \ldots, a_n : t_n)$, where: (i) *oid* is a simple term; (ii) $t_1, \ldots, t_n$ are terms; (iii) $c \in K$ is a class name, (iv) the closure of c is $\sigma_c = \{a_1 : c_1, \ldots, a_m : c_m\}$ (v) $n \leqslant m$ (i.e. n is less then or equal to the arity of c).
- A *relation atom* is an expression $r(a_1 : t_1, \ldots, a_n : t_n)$, where: (i) $t_1, \ldots, t_n$ are terms; (ii) $c \in R$ is a relation name, (iii) the scheme of r is $\Sigma_r = \langle r, \{(a_1 : c_1), \ldots, (a_m : c_m)\}\rangle$ (iv) $n \leqslant m$ (i.e. n is less than or equal to the arity of r).
- A *built-in atom* is an expression of the form $t_i \theta t_j$, where t_i and t_j are simple terms and $\theta \in \{=, <, <=, >=, >, <>\}$.
- A *derived atom* p is an expression of the form $p(t_1, \ldots, t_n)$[10] where $p \in Q$ is an derived predicate name and $t_1, \ldots, t_n$ are terms.
- An *aggregate atom*[11] is $f(S) \prec T$, where $f(S)$ is an aggregate function, S is a set term, $\prec \in \{=, <, \leqslant, >, \geqslant\}$ is a predefined comparison operator, and T is a term (variable or constant) referred to as guard.

A *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{Vars : Conj\}$, where *Vars* is a list of variables and *Conj* is a conjunction of atoms. A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj\rangle$, where $\bar{t}$ is a list of constants of D and *Conj* is a ground (variable free) conjunction of atoms.

Example 6. Consider the following expression:

$$john : person(father : person(birthplace : place(name : \text{“}Rome\text{”})), name : X).$$

We have that:

- “*Rome*” is a simple term (a string);
- *john* is a simple term ($john \in D$);
- X is a simple term (X is a variable);
- $place(name : \text{“}Rome\text{”})$ is a complex term ($place \in K$ and $|\sigma_{place}| = 1$);
- person(birthplace : place(name : “Rome”)) is a *nested* complex term ($person \in K$, $(birthplace : place) \in \sigma_{person}$ and $place(name : \text{“}Rome\text{”})$ is a term).
- $john : person(father : person(birthplace : place(name : \text{“}Rome\text{”})),\ name : X)$ is a non-ground class atom.

Definition 10. A *disjunctive* rule r is a formula:

$$a_1 \vee \cdots \vee a_n \text{ :- } b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{ not } b_m. \quad n, m \geqslant 0$$

where $a_1, \ldots, a_n$ are derived atoms, $b_1, \ldots, b_m$ are atoms, and $n \geqslant 0$, $m \geqslant k \geqslant 0$.

[10] Note that derived atoms do not have typed attributes, one can assume that their attributes have type object. Moreover, unlike for class and relation atoms, the order of terms for derived atoms is meaningful, like in standard disjunctive logic programming.

[11] For further background on aggregates see Appendix A.

We call *literal* an atom a or its negation `not` a, where `not` is the negation as failure symbol. The set $H(r) = \{a_1, \ldots, a_n\}$ is called the *head* of r, the set $B^+(r) = \{b_1, \ldots, b_k\}$ is called the *positive body* of r, the set $B^-(r) = \{b_{k+1}, \ldots, b_m\}$ is called the *negative body* of r, and $B(r) = B^+(r) \cup B^-(r)$ is called the *body* of r. If $B(r) = \emptyset$, r is called *fact*, and the :– symbol is omitted. If $H(r) = \emptyset$, r is called *constraint*. A rule r is ground if no variables appear in it.

Example 7. Consider the following disjunctive rule r:

$$color(X, \text{“}red\text{”}) \vee color(X, \text{“}blue\text{”}) \vee color(X, \text{“}green\text{”}) \text{ :– } X : nation().$$

We have that: the head of r is $H(r) = \{color(X, \text{“}red\text{”}), color(X, \text{“}blue\text{”}), color(X, \text{“}green\text{”})\}$ which contains three derived atoms; while the body of r is $B(r) = \{X : nation()\}$ which contains a class atom. Note that r contains the variable X and consequently is not ground.

Definition 11. An *axiom* a is an expression:

$$\text{:– } b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{ not } b_m. \quad m \geqslant 1$$

where $b_1, \ldots, b_m$ are atoms.

As for rules, the set $B^+(a) = \{b_1, \ldots, b_k\}$ is called the *positive body* of a, the set $B^-(a) = \{b_{k+1}, \ldots, b_m\}$ is called the *negative body* of a, and $B(a) = B^+(a) \cup B^-(a)$ is the *body* of a.

Moreover, we require that if X is a variable occurring in a, there is an atom $a_i \in B^+(a)$ such that X occurs in a_i.[12] All kinds of atoms (class atoms, relation atoms, etc.) can occur in an axiom a, apart from derived atoms, since axioms are intended to check the properties of the ontology; they do not interact with reasoning modules, which will be described later.

Example 8. Consider the following toy ontology modeling heritage monuments and their sites:

% Schemes definition

class *place*(*name* : *string*).

class *monument*(*name* : *string*).

relation *mon-site*(*mon* : *monument*, *pla* : *place*).

% Instances and tuples declarations

rome : *place*(*name* : “*Rome*”).

paris : *place*(*name* : “*Paris*”).

london : *place*(*name* : “*London*”).

colosseum : *monument*(*name* : “*Anfiteatro Flavio*”).

eiff : *monument*(*name* : “*Tour Eiffel*”).

monSite(*mon* : *colosseum*, *pla* : *rome*).

monSite(*mon* : *eiff*, *pla* : *paris*).

monSite(*mon* : *eiff*, *pla* : *london*).

where the *monSite* relation associates a monument with its site.

Now, consider the following two axioms:

(i) :– $monSite(mon : eiff, pla : X), monSite(mon : colosseum, pla : X)$.
(ii) :– $monSite(mon : M, pla : X), monSite(mon : M,\ pla : Y),\ Y \neq X$.

[12] This is, basically, the *safety* condition.

The first axiom ensures that the Tour Eiffel and the Colosseum were built in different places; the second axiom ensures that the same monument cannot be in two different places.

Note that, the given ontology is consistent w.r.t. (i) but is inconsistent w.r.t. (ii) because the Tour Eiffel appears to be both in Paris and in London.

Note that axioms are different from rules. Instead of deriving new knowledge an axiom states that a condition should be verified in an ontology (axioms are only consistency checking constructs). Axioms are equivalent to integrity constraints in disjunctive logic programming, because they are used to express a condition that should be satisfied. Intuitively, an ontology $\mathcal{O}$ is consistent w.r.t. an axiom a, if the property asserted by a is verified in $\mathcal{O}$.

We now describe the deductive part of DLP$^+$, defining *Reasoning modules* and *Queries*. First, we describe the *Reasoning modules* which allow to exploit the full power of disjunctive Datalog defining complex reasoning tasks. Then, we describe queries that can be used interrogate an ontology.

Definition 12. Given an ontology $\mathcal{O}$, a *reasoning module* R_m is a pair $R_m = \langle m, \Pi_m \rangle$ where:

- $m \in M$ is a reasoning module name,
- Π_m is a set of disjunctive rules called the definition of R_m.

We require that for each rule $r \in R_m$ the following two conditions hold:

(i) if V is a variable occurring in r then there is a literal $l \in B^+(r)$ such that V occurs in l;
(ii) if $\alpha \in H(r)$ is an atom (occurring in the head of r), and α is of the form $r(v_1, \ldots, v_n)$, then v_i is a simple term for each $i \in [1, n]$.

Intuitively, condition (i) ensures that the rules in R_m are safe (see Appendix A); while, condition (ii) ensures that complex terms do not appear in heads.

A reasoning module is syntactically declared as follows:

$$\textbf{\textit{module}}(n)\{r_1 \cdots r_n\}$$

where n is the name of the module and the definition of the module is $\Pi_n = \{r_1, \ldots, r_n\}$.

Example 9. Consider the following ontology:

class *nation* (*language* : *string*).

relation *neighbor* (*n*1 : *nation*, *n*2 : *nation*).

which models a map indicating neighbor nations, and consider the following set of instances modeling a part of the European map:

italy : *nation*(*language* : "*Italian*").

austria : *nation*(*language* : "*German*").

france : *nation*(*language* : "*French*").

germany : *nation*(*language* : "*German*").

neighbor(*n*1 : *italy*, *n*2 : *austria*).

neighbor(*n*1 : *italy*, *n*2 : *france*).

neighbor(*n*1 : *austria*, *n*2 : *germany*).

neighbor(*n*1 : *france*, *n*2 : *germany*).

The following module encodes the well-known 3-colorability problem:

module(*coloring*){

(r_1) $color(X, \text{“}red\text{”}) \vee color(X, \text{“}blue\text{”}) \vee color(X, \text{“}green\text{”}) \text{ :- } X : nation().$

(r_2) $\text{:- } neighbor(n1 : X, n2 : Y), color(X, C), color(Y, C), X \neq Y.\}$

We have that, the name of that module is *coloring* and its definition is $\Pi_{coloring} = \{r_1, r_2\}$.

The intuitive meaning of the rules is the following: the *disjunctive* rule r_1 states that every nation of the graph is colored red or yellow or green, while constraint r_2 forbids the assignment of the same color to any adjacent nodes.[13] Note that the given map is not 3-colorable.

Definition 13. Given an ontology $\mathcal{O} = \langle \mathcal{C}_s, \mathcal{R}_s, \mathcal{C}_i, \mathcal{R}_i \rangle$, and a reasoning module R_m, a query Q on $\mathcal{O}$ and R_m is a conjunction $a_1, \ldots, a_n$ of atoms.

Queries are specified in DLP$^+$ as follows:

$a_1, \ldots, a_n?$

where $a_1, \ldots, a_n$ are atoms.

Example 10. Queries can be used to interrogate an ontology, as an example, given the ontology given in Example 2 consider the following query:

$X : person(father : person(birthplace : place(name : \text{“}Rome\text{”})))?$

This query asks for the list of persons having a father who is born in Rome.

Queries can refer also to derived atoms defined in a reasoning module. Given the ontology and the reasoning module of Example 9, the following query asks for the list of nations colored in red having language “German”.

$color(nation(language : \text{“}German\text{”}), \text{“}red\text{”})?$

Note that *color* is a derived predicate defined in the reasoning module *coloring*.

4.2. From DLP$^+$ to DLP

In this section, we describe a framework containing algorithms able to “rewrite” each construct of the DLP$^+$ language (ontologies, axioms, rules, and queries) to an “equivalent” DLP program. The definition of consistent ontology, and the semantics of the reasoning modules will be given, in the following sections, in terms of the corresponding DLP programs obtained by applying the algorithms described here.

First of all we describe the functions *flatten, completeScheme*, and *orderAttributes*, which perform a sort of “normalization” of atoms and rules, removing complex terms, and adding omitted attributes to atoms. Then, we exploit these basic functions to implement *rewriteLiteral, rewriteOntology, rewriteAxiom*, and *rewriteRule* which (as intuitively suggested by their names), respectively, allow the rewriting of DLP$^+$ literals, ontologies, axioms, and reasoning modules in DLP.

The flatten *function (rewriting of complex terms)*

We now address one of the main differences between DLP atoms and DLP$^+$ atoms: the presence of complex terms.

We remove complex terms by using the function *flatten* which, given a DLP$^+$ literal $b(t_1, \ldots, t_n)$ produces an equivalent conjunction of literals. The *flatten* function is defined as follows:

- $flatten(b(t_1, \ldots, t_n)) = b(t_1, \ldots, t_n)$, if $t_1, \ldots, t_n$ are simple terms.
- $flatten(b(t_1, \ldots, t_n)) = b(u_1, \ldots, u_n), flatten(w_1), \ldots, flatten(w_m)$, where:
  - $u_i = t_i$, if t_i is a simple term
  - $u_i = V_k$, and $w_k = V_k : t_i$ if t_i is a complex term and V_k is a variable not appearing elsewhere.
- $flatten(\texttt{not } b(t_1, \ldots, t_n)) = \texttt{not } (flatten(b(t_1, \ldots, t_n)))$, where $flatten(b(t_1, \ldots, t_n))$ is a conjunction of DLP literals (which, as will be clear later, will be transformed into a new atom and an auxiliary rule).

[13] As will be more clear later, the minimality of answer sets guarantees that every node is assigned only one color.

The completeScheme *and the* orderAttributes *functions (normalization)*

Another problem we have to deal with is the labeling of terms within atoms, which allows to specify attributes without worrying about the order, and allows to omit anonymous variables.

We cannot just drop labels (attribute names), as a correspondence between terms and attributes has to be ensured, and moreover, in the case of class atoms, we have to correctly deal with oids.

Therefore, let's start by adding to each atom a the "missing" attributes by using the *completeScheme* function.

completeScheme replaces each class atom (resp. relation atom) of the form $oid : e(a_1 : t_1, \ldots, a_m : t_m)$ (resp. $e(a_1 : t_1, \ldots, a_m : t_m)$) by an equivalent one obtained by adding the missing terms as follows: $oid : e(a_1 : t_1, \ldots, a_m : t_m, a_{m+1} : \_, \ldots, a_n : \_)$ (resp. $e(a_1 : t_1, \ldots, a_m : t_m, a_{m+1} : \_, \ldots, a_n : \_)$) where $\{a_{m+1}, \ldots, a_n\} = \sigma_e \backslash \{a_1, \ldots, a_m\}$ (resp. $\{a_{m+1}, \ldots, a_n\} = A_e \backslash \{a_1, \ldots, a_m\}$)[14] are the remaining attributes in the closure σ_e of e (resp. in the scheme of e).

Then, we fix an ordering[15] for the attributes of σ_e and we use it to order the arguments of any atom by calling the *orderAttributes* function.

The rewriteLiteral *function (literal rewriting)*

We now describe how to rewrite a DLP$^+$ literal. Note that there is no direct rewriting from DLP$^+$ literals to standard DLP literals because of the presence of complex terms.[16] The *rewriteLiteral* function transforms a DLP$^+$ literal l to a pair $\langle \mathcal{R}(l), \Pi(l) \rangle$ where $\mathcal{R}(l)$ is a conjunction of DLP literals, and $\Pi(l)$ is a set of DLP rules. First, the functions: *flatten, completeScheme*, and *orderAttributes* are called on l, then $\mathcal{R}(l)$ and $\Pi(l)$ are built by the following steps:

- (i) strip off attribute names from l.
- (ii) if l is a class atom of the form $t_0 : c(t_1, \ldots, t_n)$ replace l by a temporary atom of the form $c(t_0, t_1, \ldots, t_n)$ (i.e. put the term that fills the oid's attribute in the first position).
- (iii) replace each negative literal containing a conjunction of DLP literals (obtained applying the *flatten* function) of the form `not(W)` by a new positive literal $\overline{W}$, and add the rule $\overline{W} \text{ :- } W$ to $\Pi(l)$, where $\overline{W}$ is a new DLP atom whose arguments are the variables occurring in W.
- (iv) set $\mathcal{R}(l)$ to the atom obtained executing (i), (ii), and (iii).

Note that if the literal l is positive (is an atom) then $\Pi(l)$ is empty, and if l does not contain any complex term, then $\mathcal{R}(l)$ is a DLP atom.

The rewriteOntology *function (ontology rewriting)*

This function, given an ontology $\mathcal{O} = \langle \mathcal{C}_s, \mathcal{R}_s, \mathcal{C}_i, \mathcal{R}_i \rangle$, basically, builds a set of DLP facts $\mathcal{R}(\mathcal{O})$ representing $\mathcal{O}$. This can be straightforwardly obtained (see Fig. 1) by applying the *rewriteLiteral* function to each instance (i.e. atom) in $\mathcal{C}_i$ and $\mathcal{R}_i$. Note that in this case no additional rule is generated by *rewriteLiteral* ($\Pi(a) = \emptyset$ for each atom in the extension of $\mathcal{O}$) because the extension of $\mathcal{O}$ is made of ground atoms only.

The rewriteAxiom *function (axiom rewriting)*

The rewriting of an axiom is performed, as in the case of ontology rewriting, exploiting the *rewriteLiteral* function. The function *rewriteAxiom*, reported in Fig. 2, given an axiom α, outputs a DLP program $\mathcal{R}(\alpha)$ containing a constraint (directly corresponding to α), and (possibly) a set of rules generated in order to deal with negative literals enclosing complex terms.

The rewriteRule *function (rule rewriting)*

The *rewriteRule*, reported in Fig. 3, given a rule r outputs an equivalent DLP program $\mathcal{R}(r)$ containing a rule (directly corresponding to r) and (possibly) a set of rules generated in order to deal with negative literals enclosing complex terms. Basically, $\mathcal{R}(r)$ is obtained, as for axioms, by calling the *rewriteLiteral* function for each literal l in r.

[14] Recall that σ_e is the scheme closure of e, and A_e is the set of attributes in the scheme of e.

[15] Note that the selected ordering is not relevant, but it must be the same during an entire transformation. A possible candidate is the lexicographic order between attribute names.

[16] The problem arises in presence of negative literals containing complex terms.

ALGORITHM rewriteOntology:
Input: An ontology $\mathcal{O} = \langle \mathcal{C}_s, \mathcal{R}_s, \mathcal{C}_i, \mathcal{R}_i \rangle$;
Output: The DLP program $\mathcal{R}(\mathcal{O})$.

$\mathcal{R}(\mathcal{O}) = \emptyset$
foreach class atom $I \in \mathcal{C}_i$
  $\langle \mathcal{R}(I), \Pi(I) \rangle =$ rewriteLiteral(I)
  $\mathcal{R}(\mathcal{O}) = \mathcal{R}(\mathcal{O}) \cup \mathcal{R}(I)$

foreach relation atom $T \in \mathcal{R}_i$
  $\langle \mathcal{R}(T), \Pi(T) \rangle =$ rewriteLiteral(T)
  $\mathcal{R}(\mathcal{O}) = \mathcal{R}(\mathcal{O}) \cup \mathcal{R}(T)$

Fig. 1. The *RewriteOntology* function.

ALGORITHM rewriteAxiom:
Input: An axiom α;
Output: The DLP program $\mathcal{R}(\alpha)$.
Variables: A constraint c;

$\mathcal{R}(\alpha) = \emptyset$
foreach literal $l \in \alpha$
  $\langle \mathcal{R}(l), \Pi(l) \rangle =$ rewriteLiteral(l)
  $B(c) = B(c) \cup \{\mathcal{R}(l)\}$ (add $\mathcal{R}(l)$ to the body of c)
  $\mathcal{R}(\alpha) = \mathcal{R}(\alpha) \cup \Pi(l)$

$\mathcal{R}(\alpha) = \mathcal{R}(\alpha) \cup c$

Fig. 2. The *RewriteAxiom* function.

ALGORITHM rewriteRule:
Input: A rule r;
Output: The DLP program $\mathcal{R}(r)$.
Variables: A rule $\bar{r}$;

$\mathcal{R}(r) = H(\bar{r}) = B(\bar{r}) = \emptyset$
foreach atom $a \in H(r)$
  $\langle \mathcal{R}(a), \Pi(a) \rangle =$ rewriteLiteral(a)
  $B(\bar{r}) = B(\bar{r}) \cup \{\mathcal{R}(s)\}$ (add $\mathcal{R}(s)$ to the body of $\bar{r}$)

foreach literal $l \in B(r)$
  $\langle \mathcal{R}(l), \Pi(l) \rangle =$ rewriteLiteral(l)
  $B(\bar{r}) = B(\bar{r}) \cup \{\mathcal{R}(l)\}$ (add $\mathcal{R}(l)$ to the body of $\bar{r}$)
  $\mathcal{R}(r) = \mathcal{R}(r) \cup \Pi(l)$

$\mathcal{R}(r) = \mathcal{R}(r) \cup \bar{r}$

Fig. 3. The *RewriteRule* function.

The rewriteModule *function (reasoning module rewriting)*

The rewriting of a reasoning module is obtained straightforwardly by applying the *rewriteRule* function (see Fig. 4). Basically, given a reasoning module R_m, *rewriteModule* builds the DLP program $\mathcal{R}(R_m)$ corresponding to R_m setting $\mathcal{R}(R_m) = \bigcup_{r \in R_m} \mathcal{R}(r)$.

The rewriteQuery *function (query rewriting)*

The *rewriteQuery* function (Fig. 5) rewrites a DLP$^+$ query $\mathcal{Q}$ in an equivalent DLP query $\mathcal{R}(\mathcal{Q})$. The algorithm, essentially, uses the *rewriteLiteral* function to rewrite each literal of $\mathcal{Q}$.

ALGORITHM rewriteModule:
Input: A reasoning module R_m;
Output: The DLP program $\mathcal{R}(R_m)$.

$\mathcal{R}(R_m) = \emptyset$
foreach rule $r \in R_m$
    $\mathcal{R}(R_m) = \mathcal{R}(R_m) \cup \mathcal{R}(r)$

Fig. 4. The *RewriteModule* function.

ALGORITHM rewriteQuery:
Input: A DLP$^+$ query $\mathcal{Q}$;
Output: The DLP query $\mathcal{R}(\mathcal{Q})$.

foreach literal $l \in \mathcal{Q}$
    $\langle \mathcal{R}(l), \Pi(l) \rangle$ = rewriteLiteral(l)
    add $\mathcal{R}(l)$ to $\mathcal{R}(\mathcal{Q})$

Fig. 5. The *RewriteQuery* function.

4.2.1. Consistency

We now give the formal definition of consistent ontologies in terms of the satisfiability (i.e. existence of an answer set) of a DLP program $\mathcal{P}_\alpha(\mathcal{O})$ which is obtained by applying the procedures *rewriteOntology*, and *rewriteAxiom* described in the previous section.

Definition 14. Given an ontology $\mathcal{O} = \langle \mathcal{C}_s, \mathcal{R}_s, \mathcal{C}_i, \mathcal{R}_i \rangle$ and a set of axioms $\mathcal{A}$, we say that $\mathcal{O}$ is consistent w.r.t. $\mathcal{A}$ if the program $\mathcal{P}_\alpha(\mathcal{O}) = \bigcup_{\alpha \in \mathcal{A}} \mathcal{R}(\alpha) \cup \mathcal{R}(\mathcal{O})$ has an answer set, where $\mathcal{R}(\mathcal{O})$ is the rewriting of $\mathcal{O}$ by *rewriteOntology* and $\mathcal{R}(\alpha)$ is the rewriting of all axioms in $\mathcal{A}$ by *rewriteAxiom*.

Note that the program $\mathcal{P}_\alpha(\mathcal{O})$ is stratified and non disjunctive (see Appendix A), consequently it admits at most one answer set.

Example 11. Consider the ontology and the two axioms defined in Example 8. The DLP program $\mathcal{R}(\mathcal{O})$ resulting from the application of the function *rewriteOntology* on $\mathcal{O}$ is:

place(*rome*, "*Rome*").

place(*paris*, "*Paris*").

place(*london*, "*London*").

monument(*colosseum*, "*Anfiteatro Flavio*").

monument(*eiff*, "*Tour Eiffel*").

monSite(*colosseum*, *rome*).

monSite(*eiff*, *paris*).

monSite(*eiff*, *london*).

The rewriting of axiom (i) resulting by the application of the *rewriteAxiom* function is

:– *monSite*(*eiff*, X), *monSite*(*colosseum*, X).

Note that the constraint is satisfied, and consequently the ontology is consistent.
The rewriting of axiom (ii) yields:

:– *monSite*(M, X), *monSite*(M, Y), $Y \neq X$.

In this case the constraint is violated by the assignment $M = eiff$, $X = paris$, $Y = london$, thus, the ontology is inconsistent w.r.t. (ii).

4.2.2. Brave and cautious reasoning

The semantics of a reasoning module R_m is given in terms of the semantics of an equivalent disjunctive logic program $\mathcal{R}(R_m)$. $\mathcal{R}(R_m)$ is obtained applying the function *rewriteRule* for each rule r in R_m as described in previous section.

Roughly, the idea is that the answer sets of $\mathcal{R}(R_m)$ are the semantics of R_m.[17]

Definition 15. Given an ontology $\mathcal{O} = \langle \mathcal{C}_s, \mathcal{R}_s, \mathcal{C}_i, \mathcal{R}_i \rangle$, and a reasoning module R_m, an atom A is a brave (resp. cautious) consequence of R_m and $\mathcal{O}$ if $\mathcal{R}(A)$ is a brave consequence (resp. cautious consequence) of the program $\mathcal{R}(\mathcal{O}, R_m) = \mathcal{R}(\mathcal{O}) \cup \mathcal{R}(R_m)$, where $\mathcal{R}(A)$, $\mathcal{R}(\mathcal{O})$, and $\mathcal{R}(R_m)$ are the rewriting of A, $\mathcal{O}$ and R_m, respectively.

This definition extends trivially to an arbitrary number of reasoning modules.

Example 12. Consider the ontology and the reasoning module defined in Example 9; their rewriting yields the following:

nation(*italy*, "*Italian*").

nation(*austria*, "*German*").

nation(*france*, "*French*").

nation(*germany*, "*German*").

neighbor(*italy*, *austria*).

neighbor(*italy*, *france*).

neighbor(*austria*, *germany*).

neighbor(*france*, *germany*).

$(\overline{r_1})$ $color(X,\ \text{"}red\text{"}) \vee color(X,\ \text{"}blue\text{"}) \vee color(X,\ \text{"}green\text{"}) \text{ :- } nation(X, \_).$

$(\overline{r_2})$ $\text{ :- } neighbor(Y, X), color(X, C), color(Y, C),\ X \neq Y.$

For instance, we have that $color(italy,\ \text{"}red\text{"})$ is *bravely* true; while it is cautiously false.

5. Computational complexity

In this section, we address some complexity issues related to the DLP$^+$ language. We consider the propositional case, that is, in the complexity analysis we assume that ontologies, reasoning modules, and atoms are ground (variable-free).[18] For NP-completeness and complexity theory, the reader is referred to [25].

The main decision problems arising in the framework of DLP$^+$ are the following:

- (*Consistency*) Does a DLP$^+$ ontology $\mathcal{O}$ satisfy a given set $\mathcal{A}$ of axioms?
- (*Brave Reasoning*) Given an atom A and a reasoning module R_m over a DLP$^+$ ontology $\mathcal{O}$, is A a brave consequence of R_m and $\mathcal{O}$?
- (*Cautious Reasoning*) Given an atom A and a reasoning module R_m over a DLP$^+$ ontology $\mathcal{O}$, is A a cautious consequence of R_m and $\mathcal{O}$?

The consistency of a DLP$^+$ ontology w.r.t. a given set of axioms can be efficiently checked.

Theorem 1. *Deciding whether a* DLP$^+$ *ontology* $\mathcal{O}$ *satisfies a given set* $\mathcal{A}$ *of axioms is decidable in polynomial time.*

[17] Modulo a straightforward syntactic transformation reconstructing back the DLP$^+$ atoms from the DLP atoms in the answer sets of $\mathcal{R}(R_m)$.

[18] Analyzing the complexity of the propositional case is very frequent for logic-based languages. Moreover, for DLP$^+$ this complexity essentially coincides with the data complexity [28] of non-ground programs.

Proof. To check consistency, we proceed in three steps:

(1) We transform the set $\mathcal{A}$ of DLP$^+$ axioms into a program $\mathcal{R}(\mathcal{A})$ containing integrity constraints and DLP rules by using the algorithm of Fig. 2.
(2) We build the set $\mathcal{R}(\mathcal{O})$ of logic facts representing the ontology $\mathcal{O}$.
(3) We check that $\mathcal{R}(\mathcal{A}) \cup \mathcal{R}(\mathcal{O})$ is consistent.

Step (1) and (2) are clearly doable in polynomial time. Also step (3) is tractable, because all rules in $\mathcal{R}(\mathcal{A}) \cup \mathcal{R}(\mathcal{O})$ are disjunction-free and positive (negation appears only in the integrity constraints). □

We next prove that the complexity of reasoning in DLP$^+$ is exactly the same as in traditional disjunctive logic programming. That is, objects come for free, as the addition of classes, types, inheritance, and complex objects, do not cause any computational overhead.

Theorem 2. *Brave Reasoning and Cautious Reasoning in* DLP$^+$ *are, respectively,* Σ_2^P*-complete and* Π_2^P*-complete.*

Proof. Hardness is evident, because DLP$^+$ is a superset of DLP. To prove membership, we reduce brave reasoning and cautious reasoning in DLP$^+$, to the corresponding reasoning tasks in DLP as follows.

We first build the set $\mathcal{R}(\mathcal{O})$ of logic facts representing $\mathcal{O}$. We then transform the DLP$^+$ atom A and the reasoning module R_m into the equivalent DLP conjunction $\mathcal{R}(A)$ and program $\mathcal{R}(R_m)$, respectively. (Note that the size of $\mathcal{R}(\mathcal{O})$, $\mathcal{R}(A)$, and $\mathcal{R}(R_m)$, is polynomial in the size of $\mathcal{O}$, A, and R_m, respectively.)

By Definition 15, deciding whether A is a brave/cautious consequence of R_m over $\mathcal{O}$ amounts to checking whether $\mathcal{R}(A)$ is a brave/cautious consequence of the ground DLP program $\mathcal{R}(\mathcal{O}) \cup \mathcal{R}(R_m)$. The theorem then follows from the results on the complexity of reasoning over DLP programs with aggregates shown in [4,9]. □

6. The DLV$^+$ system

DLV$^+$ is a complete tool that allows one to specify, navigate, query and perform reasoning on DLP$^+$ ontologies. We refrain describing the implementation details of DLV$^+$ in this paper. Rather, we illustrate the overall DLV$^+$ architecture, and present the main features of the system by describing the main components of the graphical user interface of DLV$^+$.

6.1. System architecture

The system architecture of DLV$^+$, depicted in Fig. 6, is composed of eight modules, namely, GUI, Parser, Data Handler, Type Checker, Intelligent Rewriter, Output Handler and Message Handler, DLV, and two libraries: Lucene and, JGraph.

The user exploits the system through an easy-to-use visual environment called GUI (Graphical User Interface). The GUI combines a number of specialized visual tools for authoring, browsing and querying a DLP$^+$ ontology. In particular, the GUI features a graph-based ontology viewer and a graphical query environment, which are based on JGraph, an open-source library.

The Parser has the job to analyze and load the content of a DLP$^+$ text file in the data structures supplied by the Data Handler. The Data Handler provides all the methods needed to access and manipulate the ontology components. In particular, data indexing and full-text search are based on the open-source library Lucene. The admissibility of an ontology is ensured by the Type Checker module which implements a number of type checking routines. The Intelligent Rewriter module translates DLP$^+$ ontologies, reasoning modules and queries to an equivalent Disjunctive Logic Program which runs on the DLV system. The Intelligent Rewriter features a number of optimization and caching techniques in order to reduce the time used by interacting with DLV. Reasoning results and possible error messages are handled by the Output Handler and by the Message Handler modules respectively, and are displayed by the user interface accordingly.

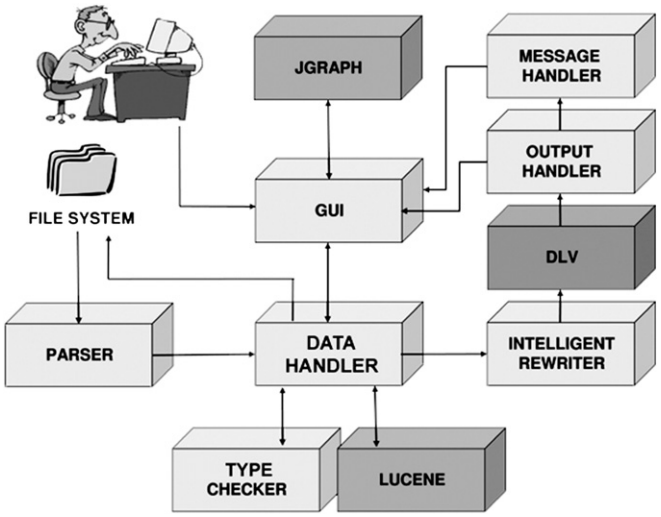

Fig. 6. The DLV$^+$ architecture.

6.2. *Implementation and usage*

The DLV$^+$ system has been implemented in Java and is based on an optimized implementation of the rewriting function described in Section 4.2.2. Moreover, the DLV$^+$ system exploits the DLV system, a state-of-the-art DLP solver that has been shown to perform efficiently on both hard and "easy" (having polynomial complexity) problems.[19]

The DLV system is a highly portable software written in ISO C++, available for various operating systems (UNIX, Mac OSX and Windows). Thus, the DLV$^+$ system runs under a variety of operating systems.

The DLV$^+$ system was designed to be simple for a novice to understand and use, and powerful enough to support experienced users. A snapshot of the system running the ontology described in Section 2 is depicted in Fig. 7. The GUI presents several panels offering access to several facilities combining the browsing environment with the editing environment.

The class/subclass hierarchy is displayed both in an indented text (on the left in Fig. 7) and a graph-based form (on the bottom in Fig. 7).

The user can browse the ontology by double-clicking the items in the panels. The structure of each ontology entity (classes, relations, and instances) can be displayed in the middle of the screen by switching between several tabbed panels. For example, in Fig. 7 the class person is selected in the class list and the class panel shows the scheme of that class. In particular, the name and the type of the class attributes are shown in a table, while, on the left, both the relations and the axioms involving the class, together with the list of the instances, are reported in an indented text form.

In the editing phase, the user enters the domain information by filling in the blanks of intuitive forms and selecting items from lists (exploiting an simple mechanism based on drag-and-drop). An up-to-date list of messages informs the user about the occurrence of errors (e.g. type checking messages, etc.) in the ontology under development. When the user clicks on an error message item the system promptly shows the entity involved in it.

Reasoning and querying can be performed by selecting the appropriate panel. Fig. 8 shows the query environment, where the user can write and execute queries. The interface also allows the reasoning modality (both brave reasoning and cautious reasoning are supported) to be selected, and the reasoning modules needed to solve the specified reasoning task to be enabled/disabled.

An important feature of the system is the visual querying interface à la QBE (depicted in Fig. 9). The user can create queries without wondering about the syntax, simply selecting classes and relations from the panels (elements can be added exploiting drag-and-drop) and creating links between class attributes and relation parameters. Importantly, the

[19] This feature is crucial for the implementation of the DLV$^+$ system, in fact ontologies are translated in an equivalent DLP program which is solved by DLV in polynomial time (under data complexity).

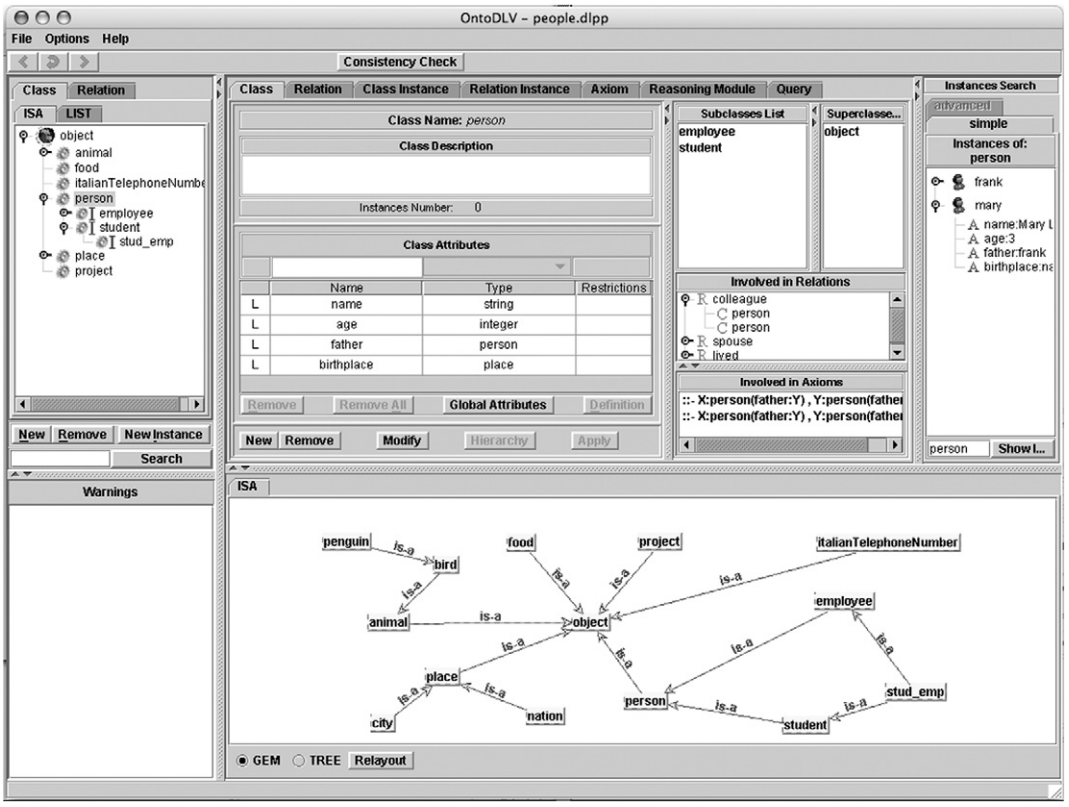

Fig. 7. DLV$^+$ GUI: Browsing and editing the ontology.

system suggests the classes or the relation that are allowed to "join" a given attribute, exploiting the strongly-typed nature of the language. Finally, a sort of "reverse-engineering" procedure allows to smoothly switch between the text editing and the visual editing environment.

Fig. 9 depicts the graphical querying environment containing the following query:

$$X : person(father : person(birthPlace : place(name : \text{“}Rome\text{”})))?$$

(i.e. who are the people whose father was born in Rome?). It is easy to see that the graphical interface makes the meaning of that query more intuitive, and it allows an unexperienced user to work with the system without knowledge about the underlying syntax details.

Finally, query results are presented to the user in an appealing way, while details about the interaction with DLV are hidden by the system.

7. Related work

A number of languages and systems somehow related to DLP$^+$ have been proposed in the literature. The most closely related system is COMPLEX [13], supporting the Complex-Datalog language, an extension of (non-disjunctive) Datalog with some concepts from the object-oriented paradigm. DLV$^+$ and COMPLEX share a similar object-oriented model, however the language of the latter is less expressive than DLP$^+$. In fact, COMPLEX supports normal (non-disjunctive) stratified programs only (its expressive power is confined to P), which are strictly less expressive than DLP$^+$ language expressing even Σ_2^P-complete properties.

Another popular logic-based object-oriented language is F-Logic [17], which includes most aspects of object-oriented and frame-based languages. F-logic was conceived as a language for intelligent information systems based on the logic programming paradigm. A main implementation of F-logic is the Flora-2 system [30] which is devoted to Semantic Web reasoning tasks. Flora-2 integrates F-Logic with other novel formalisms such as HiLog [5] (a logical

Fig. 8. DLV$^+$ GUI: Text-based interface for ontology querying.

formalism that provides higher-order and meta-programming features in a computationally tractable first-order setting) and Transaction Logic [1] (that provides a logical foundation for state changes and side effects in a logic programming language). Comparing DLP$^+$ with F-Logic, we note that the latter has a richer set of object oriented features (e.g. class methods, and multi-valued attributes), but it misses some important constructs of DLP$^+$ like disjunctive rules, which increase the knowledge modeling ability of the language. Concerning system-related aspects, an important advantage of DLV$^+$ (w.r.t. Flora-2) is the presence of a graphical development environment, which simplifies the interaction with DLV$^+$ for both the end user and the knowledge engineer.

A couple of other formalisms for specifying ontologies have been recently proposed by W3C, namely, RDF/RDFS and OWL. The Resource Description Framework (RDF) [29] is a knowledge representation language for the Semantic Web. It is a simple assertional logical language which allows for the specification of binary properties expressing that a resource (entity in the Semantic Web) is related to another entity or to a value. RDF has been extended with a basic type system; the resulting language is called RDF Vocabulary Description Language (RDF Schema or RDFS). RDFS introduces the notions of class and property, and provides mechanisms for specifying class hierarchies, property hierarchies, and for defining domains and ranges of properties. Basically, RDF(S) allows for expressing knowledge about the resources (identified via URI), and features a rich data-type library (richer than DLP$^+$), but, unlike DLP$^+$, it does not provide any way to extract new knowledge from the asserted one (RDFS does not support any "rule-based" inference mechanisms nor query facilities).

The Ontology Web Language (OWL) [27] is an ontology representation language built on top of RDFS. The ontologies defined in this language consist of *concepts* (or classes) and *roles*(binary relations also called class properties). OWL has a logic based semantics, and in general allows to express complex statements about the domain of discourse (OWL is undecidable in general) [27]. The largest decidable subset of OWL, called OWL-DL, coincides, basically, with $SHOIN(D)$, an expressive Description Logic (DL) [2]. OWL is based on classical logic (there is a direct mapping from *SHOIN* to First Order Logic (FOL)) and, consequently, is quite different form DLP$^+$, which is based on DLP. Compared to DLP$^+$, OWL misses, for instance, *default negation*, *nonmonotonic disjunction*, and

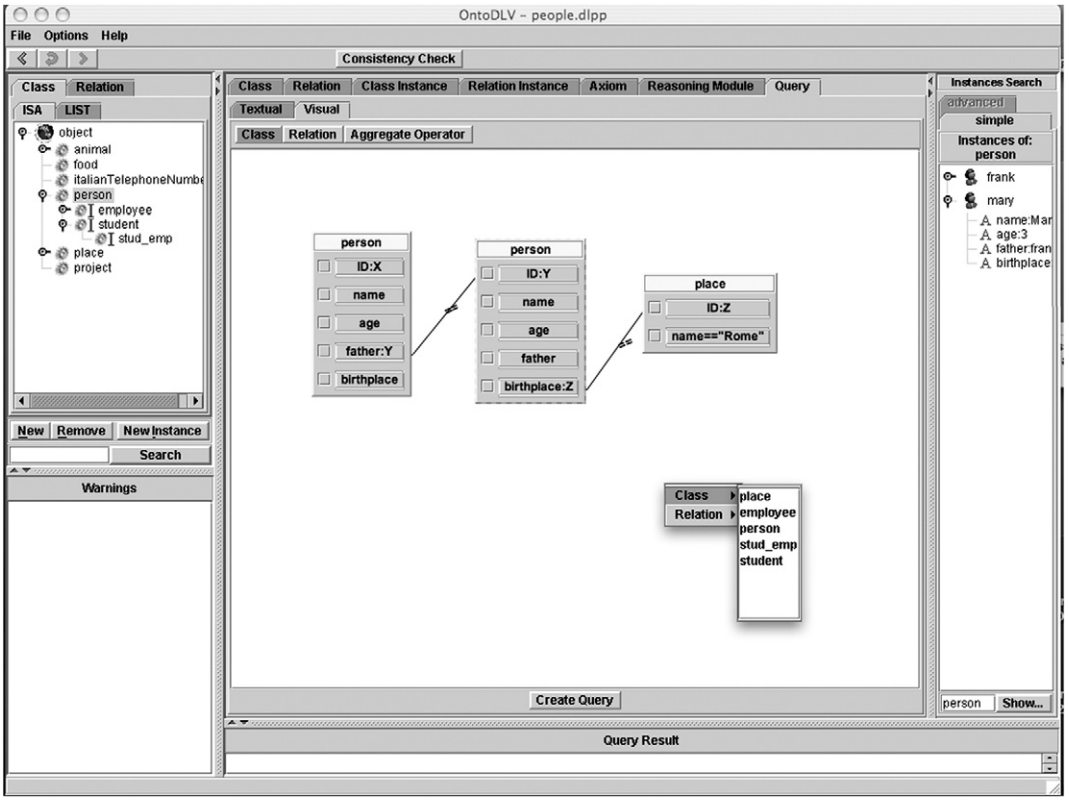

Fig. 9. DLV$^+$ GUI: The QBE-Like query interface.

inference rules. "Rules", in particular, are considered an indispensable tool for enabling agents to reason about the knowledge represented in an ontology [15,27].

Recently, B.N. Grosof et al. proposed a new logic-based language that tries to combine logic programming and DL-based languages in the so-called Description Logic Programs [14]. This approach is however limited to definite (Horn) programs (which are "common" to both FOL and logic programming). Thus, they fail in handling expressive DLs, and miss relevant DLP features (which are, obviously, present in DLP$^+$), like the *negation as failure* which cannot be expressed in FOL.

In sum, the strong point of DLP$^+$, w.r.t. to other ontology representation languages, is the natural way in which it combines the most common ontology definition constructs with a powerful logic programming language, including rules, nonmonotonic disjunction, and default negation.

Concluding, we observe that even if DLP$^+$ misses some object-oriented features, like multi-valued attributes, and intentional classes, our language represents an important and profitable extension of DLP for ontology representation and reasoning. Similar languages and systems have been quite successful and positively accepted in the literature (see e.g. [6,13,21,23]), even if were based on less powerful logic programming languages. Moreover, some of the missing features can be easily simulated in DLP$^+$ by exploiting the existing ones.[20]

8. Conclusion

In this paper, we have presented the DLP$^+$ language, an extension of disjunctive logic programming with relevant object-oriented constructs, including classes, objects, (multiple) inheritance, and types. We have formally defined the semantics of the language, and shown its usage for ontology representation and reasoning.

[20] Multi-valued and optional attributes (with cardinality 0:n and 0:1, respectively, from an E/R model point of view) can be simulated by suitable binary relations; while intentional classes can be implemented through reasoning modules.

Importantly, we have provided also a concrete implementation of DLP$^+$: the DLV$^+$ system. DLV$^+$ is built on top of DLV (the state-of-the art DLP system). It implements all features of DLP$^+$, it also provides an advanced visual-interface, and a powerful type-checking mechanism, supporting the user for fast ontologies specification and errors detection.

The DLV$^+$ system is a valid support for the development of knowledge-based applications. Indeed, even if DLP$^+$ has been released very recently, it is already employed, playing a central role, in a couple of advanced applications for information extraction and text classification: HiLEx [26] and OLEX [7,8].

Ongoing work concerns the enhancement of DLV$^+$ by extending its language with new features such as optional and multi-valued attributes, intentional classes, and a more reusable form of reasoning modules.

Acknowledgements

This work was partially supported by M.I.U.R. under projects "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione", and "ONTO-DLV: Un ambiente basato sulla Programmazione Logica Disgiuntiva per il trattamento di Ontologie" n.2521.

Appendix A. Disjunctive logic programs with aggregates

A.1. Syntax

We assume that the reader is familiar with standard DLP; we refer to atoms, literals, rules, and programs of DLP, as *standard atoms, standard literals, standard rules*, and *standard programs*, respectively. Two literals are said to be complementary if they are of the form p and `not` p for some atom p. Given a literal L, $\neg.L$ denotes its complementary literal. Accordingly, given a set A of literals, $\neg.A$ denotes the set $\{\neg.L \mid L \in A\}$. For further background, see [3,12].

Set terms. A (DLP) *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{Vars : Conj\}$, where *Vars* is a list of variables and *Conj* is a conjunction of standard atoms.[21] A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj\rangle$, where $\bar{t}$ is a list of constants and *Conj* is a ground (variable free) conjunction of standard atoms.

Aggregate functions. An *aggregate function* is of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant.

Example 13. (In the examples, we adopt the syntax of DLV to denote aggregates.) Aggregate functions currently supported by the DLV system are: `#count` (*number of terms*), `#sum` (*sum of non-negative*), `#times` (*product of positive integers*), `#min` (*minimum term*, *undefined for empty set*), `#max` (*maximum term*, *undefined for empty set*).[22]

Aggregate literals. An *aggregate atom* is $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{=, <, \leqslant, >, \geqslant\}$ is a predefined comparison operator, and T is a term (variable or constant) referred to as guard.

Example 14. The following aggregate atoms in DLV notation, where the latter contains a ground set and could be a ground instance of the former:

$$\texttt{\#max}\{Z : r(Z), a(Z, V)\} > Y.$$

$$\texttt{\#max}\{\langle 2 : r(2), a(2, k)\rangle, \langle 2 : r(2), a(2, c)\rangle\} > 1.$$

An *atom* is either a standard (DLP) atom or an aggregate atom. A *literal* L is an atom A or an atom A preceded by the default negation symbol `not`; if A is an aggregate atom, L is an *aggregate literal*.

[21] Intuitively, a symbolic set $\{X : a(X, Y), p(Y)\}$ stands for the set of X-values making $a(X, Y)$, $p(Y)$ true, i.e., $\{X \mid \exists Y$ s.t. $a(X, Y)$, $p(Y)$ is true$\}$.

[22] The first two aggregates correspond, respectively, to the cardinality and weight constraint literals of Smodels.

DLP *programs*. A (DLP) *rule* R is a construct

$$a_1 \vee \cdots \vee a_n \text{ :- } b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{ not } b_m.$$

where $a_1, \ldots, a_n$ are standard atoms, $b_1, \ldots, b_m$ are atoms, $n \geqslant 0$, and $m \geqslant k \geqslant 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is referred to as the *head* of R while the conjunction $b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{not } b_m$ is the *body* of R. We denote the set $\{a_1, \ldots, a_n\}$ of the head atoms by $H(R)$, and the set $\{b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{not } b_m\}$ of the body literals by $B(R)$.

A (DLP) *program* is a set of DLP rules. A *global* variable of a rule r is a variable appearing in a standard atom of r; all other variables are *local* variables.

Safety. A rule r is *safe* if the following conditions hold: (i) each global variable of r appears in a positive standard literal in the body of r; (ii) each local variable of r appearing in a symbolic set $\{Vars : Conj\}$ appears in an atom of $Conj$; (iii) each guard of an aggregate atom of r is a constant or a global variable. A program $\mathcal{P}$ is safe if all $R \in \mathcal{P}$ are safe. In the following we assume that DLP programs are safe.

A.2. Answer set semantics

Universe and base. Given a DLP program $\mathcal{P}$, let $U_\mathcal{P}$ denote the set of constants appearing in $\mathcal{P}$, and $B_\mathcal{P}$ be the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_\mathcal{P}$. Given a set X, let $\bar{2}^X$ denote the set of all multisets over elements from X. Without loss of generality, we assume that aggregate functions map to I (the set of integers).

Instantiation. A *substitution* is a mapping from a set of variables to $U_\mathcal{P}$. A substitution from the set of global variables of a rule r (to $U_\mathcal{P}$) is a *global substitution for* r; a substitution from the set of local variables of a symbolic set S (to $U_\mathcal{P}$) is a *local substitution for* S. Given a symbolic set without global variables $S = \{Vars : Conj\}$, the *instantiation of* S is the following ground set of pairs $inst(S)$: $\{\langle \gamma(Vars) : \gamma(Conj)\rangle \mid \gamma \ is a local substitution for \ S\}$. [23]

A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Example 15. Consider the following program $\mathcal{P}_1$:

$$q(1) \vee p(2,2). \qquad q(2) \vee p(2,1).$$
$$t(X) \text{ :- } q(X), \#\texttt{sum}\{Y : p(X,Y)\} > 1.$$

The instantiation $Ground(\mathcal{P}_1)$ is the following:

$$q(1) \vee p(2,2).t(1) \text{ :- } q(1), \#\texttt{sum}\{\langle 1 : p(1,1)\rangle, \langle 2 : p(1,2)\rangle\} > 1.$$
$$q(2) \vee p(2,1).t(2) \text{ :- } q(2), \#\texttt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1.$$

Interpretations. An *interpretation* for a DLP program $\mathcal{P}$ is a consistent set of standard ground atoms, that is $I \subseteq B_\mathcal{P}$. A positive literal A is true w.r.t. I if $A \in I$, is false otherwise. A negative literal $\texttt{not } A$ i true w.r.t. I, if $A \notin A$, is false otherwise.

An interpretation also provides a meaning for aggregate literals.

Let I be an interpretation. A standard ground conjunction is true (resp. false) w.r.t. I if all its literals are true. The meaning of a set, an aggregate function, and an aggregate atom under an interpretation, is a multiset, a value, and a truth-value, respectively. Let $f(S)$ be a an aggregate function. The valuation $I(S)$ of S w.r.t. I is the multiset of the first constant of the elements in S whose conjunction is true w.r.t. I. More precisely, let $I(S)$ denote the multiset $[t_1 \mid \langle t_1, \ldots, t_n : Conj\rangle \in S \wedge Conj$ is true w.r.t. $I]$. The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \perp$ (where $\perp$ is a fixed symbol not occurring in $\mathcal{P}$).[24]

[23] Given a substitution σ and a DLP object Obj (rule, set, etc.), we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

[24] In this paper, we assume that the value of an aggregate function can be computed in time polynomial in the size of the input multiset.

An instantiated aggregate atom $A = f(S) \prec k$ is *true w.r.t. I* if: (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ holds; otherwise, A is false. An instantiated aggregate literal $\texttt{not}\ A = \texttt{not}\ f(S) \prec k$ is *true w.r.t. I* if (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ does not hold; otherwise, A is false.

Minimal models. Given an interpretation I, a rule r is *satisfied w.r.t. I* if some head atom is true w.r.t. I whenever all body literals are true w.r.t. I. An interpretation M is a *model* of a DLP program $\mathcal{P}$ if all $R \in \mathit{Ground}(\mathcal{P})$ are satisfied w.r.t. M. A model M for $\mathcal{P}$ is (subset) minimal if no model N for $\mathcal{P}$ exists such that $N \subset M$.

Answer sets. We now recall the generalization of the Gelfond–Lifschitz transformation to programs with aggregates from [11].

Definition 16. [11] Given a ground DLP program $\mathcal{P}$ and a total interpretation I, let $\mathcal{P}^I$ denote the transformed program obtained from $\mathcal{P}$ by deleting all rules in which a body literal is false w.r.t. I. I is an answer set of a program $\mathcal{P}$ if it is a minimal model of $\mathit{Ground}(\mathcal{P})^I$.

Example 16. Consider the following two programs:

$$P_1 : \{p(a) \text{ :- } \#\texttt{count}\{X : p(X)\} > 0\}.$$

$$P_2 : \{p(a) \text{ :- } \#\texttt{count}\{X : p(X)\} < 1\}.$$

$\mathit{Ground}(P_1) = \{p(a) \text{ :- } \#\texttt{count}\{\langle a : p(a)\rangle\} > 0.\}$ and $\mathit{Ground}(P_2) = \{p(a) \text{ :- } \#\texttt{count}\{\langle a : p(a)\rangle\} < 1.\}$, and interpretation $I_1 = \{p(a)\}$, $I_2 = \emptyset$. Then, $\mathit{Ground}(P_1)^{I_1} = \mathit{Ground}(P_1)$, $\mathit{Ground}(P_1)^{I_2} = \emptyset$, and $\mathit{Ground}(P_2)^{I_1} = \emptyset$, $\mathit{Ground}(P_2)^{I_2} = \mathit{Ground}(P_2)$ hold.

I_2 is the only answer set of P_1 (because I_1 is not a minimal model of $\mathit{Ground}(P_1)^{I_1}$), while P_2 admits no answer set (I_1 is not a minimal model of $\mathit{Ground}(P_2)^{I_1}$, and I_2 is not a model of $\mathit{Ground}(P_2) = \mathit{Ground}(P_2)^{I_2}$).

Note that any answer set A of $\mathcal{P}$ is also a model of $\mathcal{P}$ because $\mathit{Ground}(\mathcal{P})^A \subseteq \mathit{Ground}(\mathcal{P})$, and rules in $\mathit{Ground}(\mathcal{P}) - \mathit{Ground}(\mathcal{P})^A$ are satisfied w.r.t. A.

References

[1] J.J. Alferes, J.A. Leite, L.M. Pereira, H. Przymusinska, T.C. Przymusinski, HiLog: A foundation for higher-order logic programming, JLP 15 (3) (1993) 187–230.

[2] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, CUP, 2003.

[3] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, CUP, 2002.

[4] F. Calimeri, W. Faber, N. Leone, S. Perri, Declarative and computational properties of logic programs with aggregates, in: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05), 2005, pp. 406–411.

[5] W. Chen, M. Kifer, D. Scott Warren, Hilog: A foundation for higher-order logic programming, JLP 15 (1993) 187–230.

[6] W. Chen, D. Scott Warren, C-logic of complex objects, in: Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 29–31, 1989, Philadelphia, PA, ACM Press, 1989, pp. 369–378.

[7] C. Cumbo, S. Iiritano, P. Rullo, Reasoning-based knowledge extraction for text classification, in: Discovery Science, 2004, pp. 380–387.

[8] R. Curia, M. Ettorre, S. Iiritano, P. Rullo, Textual document per-processing and feature extraction in OLEX, in: Proceedings of Data Mining 2005, Skiathos, Greece, 2005.

[9] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, G. Pfeifer, Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV, in: IJCAI 2003, Acapulco, Mexico, 2003, pp. 847–852.

[10] T. Eiter, W. Faber, N. Leone, G. Pfeifer, Declarative problem-solving using the DLV system, in: Logic-Based Artificial Intelligence, 2000, pp. 79–103.

[11] W. Faber, N. Leone, G. Pfeifer, Recursive aggregates in disjunctive logic programs: Semantics and complexity, in: JELIA 2004, 2004, pp. 200–212.

[12] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, NGC 9 (1991) 365–385.

[13] S. Greco, N. Leone, P. Rullo, COMPLEX: An object-oriented logic programming system, IEEE TKDE 4 (4) (1992).

[14] B.N. Grosof, I. Horrocks, R. Volz, S. Decker, Description logic programs: Combining logic programs with description logics, in: Proceedings of the Twelfth International World Wide Web Conference, WWW2003, Budapest, Hungary, 2003, pp. 48–57.

[15] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, Swrl: A semantic web rule language combining owl and ruleml, W3C Submission, http://www.w3.org/Submission/SWRL/.

[16] T. Janhunen, I. Niemelä, Gnt—a solver for disjunctive logic programs, in: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7), Fort Lauderdale, FL, 2004, pp. 331–335.

[17] M. Kifer, G. Lausen, J. Wu, Logical foundations of object-oriented and frame-based languages, JACM 42 (4) (1995) 741–843.

[18] N. Leone, G. Gottlob, R. Rosati, T. Eiter, W. Faber, M. Fink, G. Greco, G. Ianni, E. Kałka, D. Lembo, M. Lenzerini, V. Lio, B. Nowicki, M. Ruzzi, W. Staniszkis, G. Terracina, The INFOMIX system for advanced integration of incomplete and inconsistent data, in: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005), Baltimore, MA, ACM Press, 2005, pp. 915–917.
[19] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM TOCL, 2006, in press.
[20] Y. Lierler, Cmodels for tight disjunctive logic programs, in: W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany, Ulmer Informatik-Berichte, Universität Ulm, Germany, 2005, pp. 163–166.
[21] M. Liu, G. Dobbie, T. Wang Ling, Logical foundation for deductive object-oriented databases, ACM Trans. Database Syst. 27 (2002) 117–151.
[22] J. Lobo, J. Minker, A. Rajasekar, Foundations of Disjunctive Logic Programming, MIT Press, Cambridge, MA, 1992.
[23] M. Maier (Ed.), A Logic for Objects, 1986.
[24] F. Massacci, Computer aided security requirements engineering with ASP non-monotonic reasoning, ASP and Constraints, Seminar N 05171, Dagstuhl Seminar on Nonmonotonic Reasoning, Answer Set Programming and Constraints, 2005.
[25] C. Papadimitriou, Computational Complexity, 1994.
[26] M. Ruffolo, N. Leone, M. Manna, D. Sacca', A. Zavatto, Exploiting ASP for semantic information extraction, in: Proceedings ASP05—Answer Set Programming: Advances in Theory and Implementation, Bath, UK, 2005.
[27] M.K. Smith, C. Welty, D.L. McGuinness, OWL web ontology language guide, W3C Candidate Recommendation, 2003, http://www.w3.org/TR/owl-guide/.
[28] M.Y. Vardi, Complexity of relational query languages, in: Proceedings of the 14th Symposium on Theory of Computation (STOC), 1982, pp. 137–146.
[29] W3C, The resource description framework, http://www.w3.org/RDF/./.
[30] G. Yang, M. Kifer, C. Zhao, Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web, in: CoopIS/DOA/ODBASE, 2003, pp. 671–688.